DS-GA 1004 Final Project Report

Zian Jiang (**zj444**)

May 9, 2020

Introduction

This report will be mainly composed of two sections. The first section will be a detailed description of the baseline ALS model pipeline, including implementation decisions and details with results from training and evaluation. The second section will be on the visualization extension, including latent factors extraction, genre tags extraction, and visualization. The source code can be found under the `src/` directory, and the visualization extension can be found at `/addon.ipynb`.

1 Baseline model implementation and details

1.1 Downsampling

Because of scalability issues during the evaluation phase, all models will be trained on using only 10% of all interactions, and downsampling is needed. It is important to note that downsampling here is not simply sampling 10% of the `interactions` dataframe. Instead, we need to sample 10% of users, and collect all the relevant interactions of those 10% users.

First, we need to collect a list of unique users and randomly select 10% of that list as the sampled users.This can be done in two steps:

```
all_users = spark.sql("SELECT DISTINCT user_id FROM
    interactions");
sampled_users = all_users.sample(withReplacement=False
    , fraction=0.1, seed=42);
```

Given `sampled_users`, we can collect all the relevant interactions into a new dataframe.

```
sampled_interactions = spark.sql("SELECT * FROM
    interactions WHERE user_id in (SELECT * FROM
    sampled_users)")
```

Also, we decide that users with fewer than 10 interactions do not provide sufficient data for evaluation, who are thus discarded. We use the `GROUP BY` clause to count each user's interactions and filter based on the count.

```
count_interactions = spark.sql("SELECT user_id, COUNT(
    user_id) as count FROM sampled_interactions GROUP
    BY user_id")
filtered_interactions = spark.sql("SELECT interactions
    .* FROM interactions JOIN count_interactions ON
    count_interactions.user_id = interactions.user_id
    WHERE count >= 10")
```

1.2 Train/validation/test set split

We use a 60/20/20 split for train/validation/test set split. Similarly the sampling is on users instead of interactions. The procedure is slightly different from Section 1.1 since the model can't predict items for a user with no history. This requires extra steps in validation and test set generation. After following the sampling procedure in Section 1.1 to generate train/validation/test set, we need to further split interactions **of each user** by half in validation and test set and add those interactions back to the train set, so that in the training phase we have history for users in the validation and test set. Here is a code snippet that shows how to sample half of the interactions **of each user** from the validation set using `sampleBy`. Same operation can be applied to the test set.

```
val_users_list = [row["user_id"] for row in val_users.
    collect()]
val_fractions = dict(zip(val_users_list, [0.5 for _ in
     range(len(val_users_list))]))
val_training = val_interactions.sampleBy("user_id",
    fractions=val_fractions, seed=42)
val_validation = val_interactions.subtract(
    val_training)
```

1.3 Training and hyperparameters tuning

There are two hyperparameters to tune: `rank` and regularization term λ. For grid search, we will try `rank` in $\{10, 15, 20\}$ and λ in $\{0.01, 0.1, 1\}$. Also, to ensure we don't get NaN evaluation metrics, we set cold start strategy to 'drop'.

1.4 Evaluation on validation and test set

Besides `RMSE`, we also use ranking metrics such as `PrecisionAtk` ($p(k)$), `MeanAveragePrecision` (MAP) and `NormalizedDiscountedCumulativeGainAtk` (NDCG at k) for $k = 500$. While `RMSE` is very easy and fast to evaluate, one bottleneck we encounter is that the Spark `RankingMetrics()` object only takes in RDD as input and converting a dataframe into RDD on Spark takes a long time. Thus, we decide to, during grid search, use `RMSE` as the only evaluation metric and select the model with the lowest `RMSE` on the validation set. Then we evaluate the best

		λ		
		0.01	0.1	1
rank	10	1.149	1.183	1.849
	15	1.214	1.258	1.849
	20	**1.083**	1.134	1.849

Table 1: Grid search results with RMSE on the validation set. The best model achieves RMSE of 1.083 with rank 20 and $\lambda = 0.01$.

	metrics			
	RMSE	$p(500)$	MAP	NDCG at 500
the best model	1.076	0.004	0.001	0.006

Table 2: the best model with rank 20 and $\lambda = 0.01$ evaluation on the test set.

model on the test set using all ranking metrics, saving us significant time in computation.

2 Evaluation results

From Table 1, we can see that for a fixed rank, the error becomes larger as regularization increases, which confirms with our intuitions. Also, the error decreases as rank increases, as the model learns a richer latent representation with higher ranks. However, it seems when regularization term becomes too big ($\lambda = 1$), increasing rank will not improve the error anymore. From Table 2, we can see that even with a low regularization ($\lambda = 0.01$), the model does not seem to overfit at all as it achieves the lowest `RMSE` (1.083) among higher $\lambda's$.

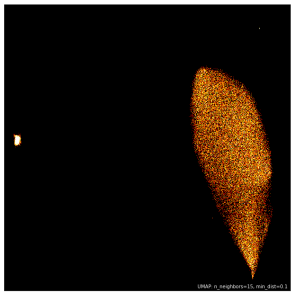
3 Visualization extension

3.1 Implementation

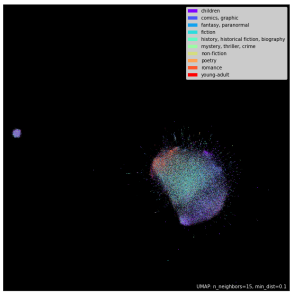
For latent item factors, we want to plot the learned representation combined with the genre tag of each book. However, the genre information we have has multiple genres for each item, generated through a keyword matching process. For example,

```
{‘book_id’: ‘7327624’,
 ‘genres’: {‘fantasy, paranormal’: 31,
            ‘fiction’: 8,
            ‘mystery, thriller, crime’: 1,
            ‘poetry’: 1}}.
```

Thus, for each item, we need a mapping function to extract the key whose value is the largest as the its genre. Below is the code snippet we use to extract genre.



(a) 200,000 latent user factors



(b) 200,000 latent item factors grouped by genres

Figure 1: UMAP visualizations of ALS model with rank 20 and $\lambda = 0.01$ trained on 100% of interactions. Both UMAPs are only trained on 200,000 samples to preserve memory.

```
def find_genre(dic):
    for k, v in dic.items():
        if v is None:
            dic[k] = 0
    return max(dic, key = dic.get)
```

Also, we train UMAP using the default parameters. Since our input matrix is of shape $(N, 20)$ which UMAP requires to store in memory for matrix operation, we find it challenging memory-wise for $N > 200,000$. Thus, we will only train UMAP after sampling out 200,000 latent user/item factors.

3.2 Visualizations

As Figure 1(b) shows, latent item factors form clusters according to their associated genres, which shows that the ALS model has successfully learned a representation that incorporates general information of the items, such as genre, to the latent space. However, the clusters are not separating. We speculate this is due to the fact that each item has multiple predicted genres, thus they are not separable.